# Clab-2 Report

ENGN6528

Yifan Zhu
u7560434

20/03/2023

# Task 1: Harris Corner Detector (6 marks)

1. **Read and understand the corner detection code 'harris.m' in Fig 1.**

2. **Complete the missing parts, rewrite 'harris.m' (or harris.py) into a Matlab (or Python) function, and design appropriate function signature (1 mark).**

In order to detect corners using the Harris corner detector, firstly we calculate R-value using the following Formula 1.1.

$$R = \det M - k(\text{trace } M)^2 \tag{1.1}$$

Where M is a 2x2 matrix computed from image derivatives shown in Formula 1.2.

$$M = \sum w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix} \tag{1.2}$$

The code for calculating R-value is shown in Figure 1.1.

```
######################################################################
# Task: Compute the Harris Corner
######################################################################

# Given an image, this function is intended to calculate
# the corner response R value of this image
# input:
#   img: image which need to be calculated;
#   k: empirical constant, default is 0.01
# output:
#   R value
def R_value(img, k=0.01):
    # Calculate the determinant and trace of matrix M
    # Derivative masks
    dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    dy = dx.transpose()

    # computer x and y derivatives of image
    Ix = conv2(img, dx)
    Iy = conv2(img, dy)

    # generate a gaussian kernel. Using default shape 3x3 and sigma 0.5
    g = fspecial()
    Iy2 = conv2(np.power(Iy, 2), g)
    Ix2 = conv2(np.power(Ix, 2), g)
    Ixy = conv2(Ix * Iy, g)

    # determinant
    det = np.multiply(Ix2, Iy2) - np.power(Ixy, 2)
    # trace
    trace = Ix2 + Iy2
```

```
    # Calculate corner response R
    R = det - k * np.power(trace, 2)
    return R
```

Figure 1.1 Code for calculating R-value

Then based on the R-valued gotten before, we perform non-maximum suppression and thresholding to get the corner points. Non-maximum suppression method is implemented by the method *non_max_sup*(), which will return the max R-value in a certain size area. Threshold method is implemented by *thresholding*(). The threshold is calculated by the multiplication of the threshold ratio and maximum R-value in a certain area. Then choose the points with an R-value greater than the threshold value and the max value in a certain area around them. These two functions are shown in Figure 1.2.

```
######################################################################
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
######################################################################

# Given an array of R value and a coordinate, find the maximum R value among
the neighbour of this coordinate
# input:
#   R: array of R values;
#   x: coordinate x
#   y: coordinate y
#   sup_size: searching size of the neighbours, default size is 3x3 as the
centre of (x, y)
# output:
#   max value of the given area
def non_max_sup(R, x, y, sup_size=3):
    a = max(0, x - (sup_size // 2))
    b = min(R.shape[0] - 1, x + (sup_size // 2)) + 1
    c = max(0, y - (sup_size // 2))
    d = min(R.shape[0] - 1, y + (sup_size // 2)) + 1
    neighbours = R[max(0, x - (sup_size // 2)):min(R.shape[0] - 1, x +
(sup_size // 2)) + 1,
                   max(0, y - (sup_size // 2)):min(R.shape[1] - 1, y +
(sup_size // 2)) + 1]
    return neighbours.max()


# Find R values which are exceeding the threshold, and apply non-maximum
method to find a local maximum value
# input:
#   img: image which need to find a corner points
#   threshold_ratio: It is used to multiply the max R value. Default is to
keep values more than 0.01*max.
#   sup_size: the size to apply non-maximum method
#   k: empirical constant to calculate R value, default is 0.01
# output:
#   the N corner points as an Nx2 matrix of x and y coordinates
def thresholding(img, threshold_ratio=0.01, sup_size=3, k=0.01):
    R = R_value(img, k)
    threshold = threshold_ratio * R.max()
    res = []
    for x in range(0, R.shape[0]):
```

```
        for y in range(0, R.shape[1]):
            if R[x, y] > threshold:
                if R[x, y] == non_max_sup(R, x, y, sup_size):
                    res.append([x, y])
    return res
```

Figure 1.2 Code for finding corner points

3. **Please provide comments on line … and every line of your solution after line … (0.5 mark). Specifically, you need to provide short comments on your code, which should make your code readable.**

Comments for Block #5 are shown in Figure 1.3.

```
# generate a gaussian filter. In the case below, sigma = 2, size of the kernel is 13 x 13
g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma)
```

Figure 1.3 Comments for Block #5

Other codes and comments after Block #7 are included in Figure 1.1 and Figure 1.2.

4. **Test this function on the provided four test images (Harris-[1,2,3,4].jpg, they can be downloaded from Wattle). Display your results by marking the detected corners on the input images (using circles or crosses, etc) (0.5 mark for each image, (2 marks) in total).**

   **Please make sure that your code can be run successfully on a local machine and generate results. If your submitted code cannot replicate your results, you may need to explain and demonstrate the results in person to tutors.**

Apply my Harris Corner Detector on Harris-1.jpg, with threshold ratio 0.001, maximum suppress size 7x7 and empirical constant k 0.04. Result is shown in Figure 1.3.

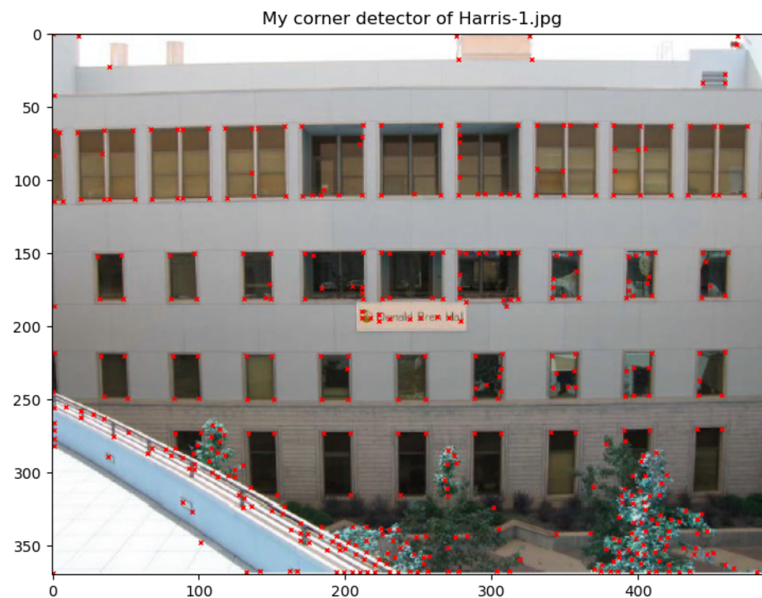



Figure 1.3 Detected corners of "Harris-1.jpg"

Apply my Harris Corner Detector on Harris-2.jpg, with threshold ratio 0.01, maximum suppress size 9x9 and empirical constant k 0.04. Result is shown in Figure 1.4.

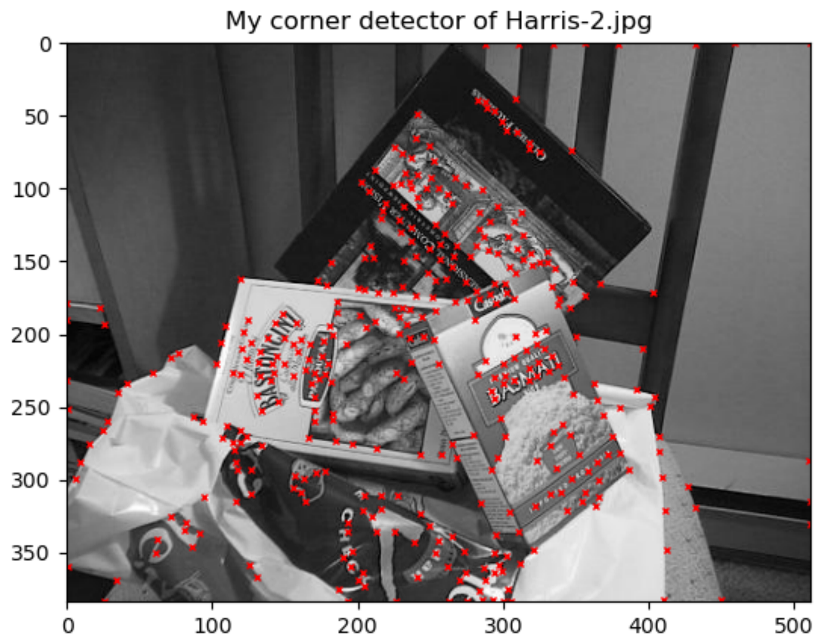



Figure1.4 Detected corners of “Harris-2.jpg”

Apply my Harris Corner Detector on Harris-3.jpg, with threshold ratio 0.005, maximum suppress size 9x9 and empirical constant k 0.04. Result is shown in Figure 1.5.

Figure 1.5 Detected corners of “Harris-3.jpg”

Apply my Harris Corner Detector on Harris-4.jpg, with threshold ratio 0.005, maximum suppress size 7x7 and empirical constant k 0.04. Result is shown in Figure 1.5.

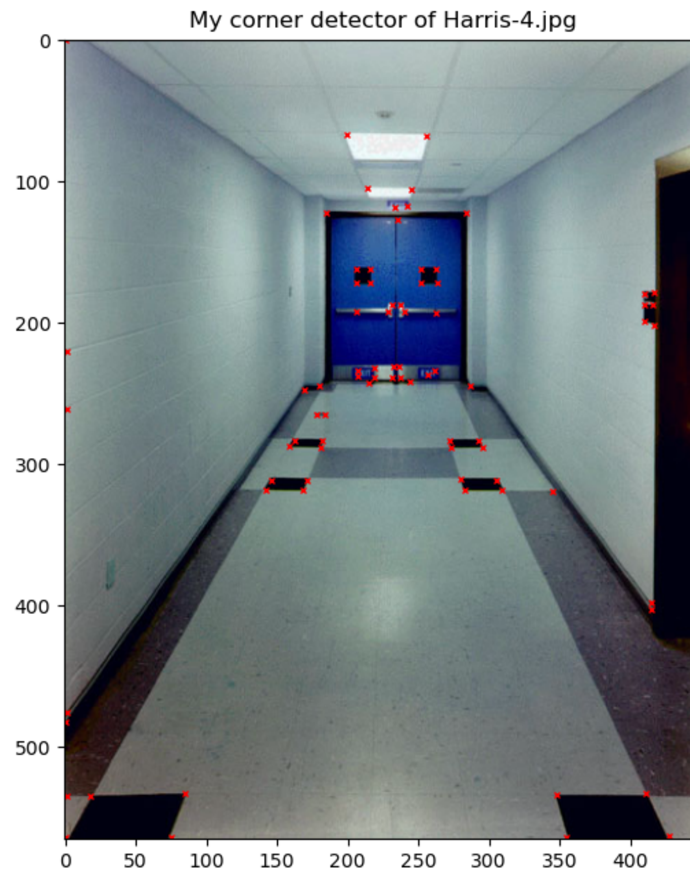
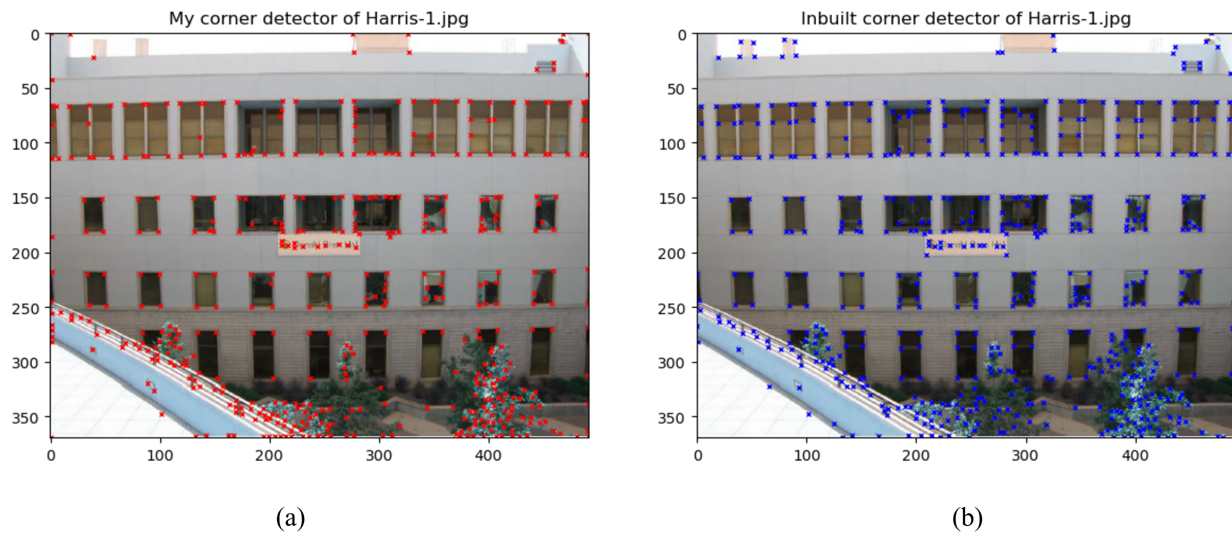



Figure 1.6 Detected corners of "Harris-4.jpg"

5. **Compare your results with that from built-in function used to detect corcers (0.5 mark), and discuss the factors that affect the performance of Harris corner detection (1 mark) .**

   **In your Lab Report, you need to list your complete source code with detailed comments and show corner detection results and their comparisons for each of the test images.**

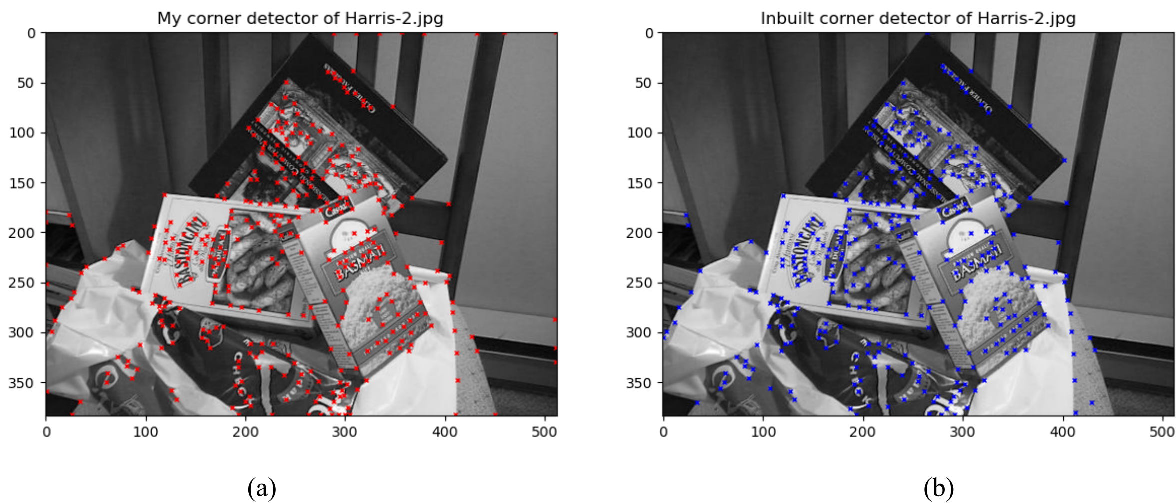
Apply both my own Harris corner detector and inbuilt Harris corner detector function on "Harris-1.jpg". The parameters for my detector are the same as described in the last question. For inbuilt function, block size is 3, k size is 3, empirical constant k is 0.04, threshold 0.001 and window for maximum suppression is 7x7. Results are shown in 1.7.




(a) (b)

Figure 1.7 Detected corners of "Harris-1.jpg". (a) My Harris corner detector (b) Inbuilt Harris corner detector

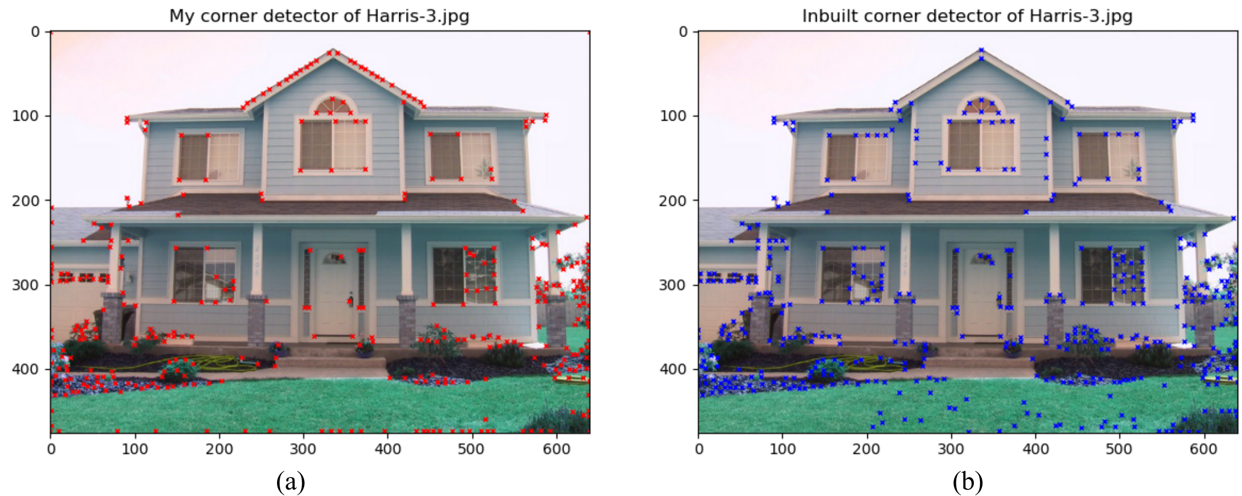
Apply both my own Harris corner detector and inbuilt Harris corner detector function on "Harris-2.jpg". The parameters for my detector are the same as described in the last question. For inbuilt function, block size is 3, k size is 3, empirical constant k is 0.04, threshold 0.01 and window for maximum suppression is 9x9. Results are shown in 1.8.




(a) (b)

Figure 1.8 Detected corners of "Harris-2.jpg". (a) My Harris corner detector (b) Inbuilt Harris corner detector

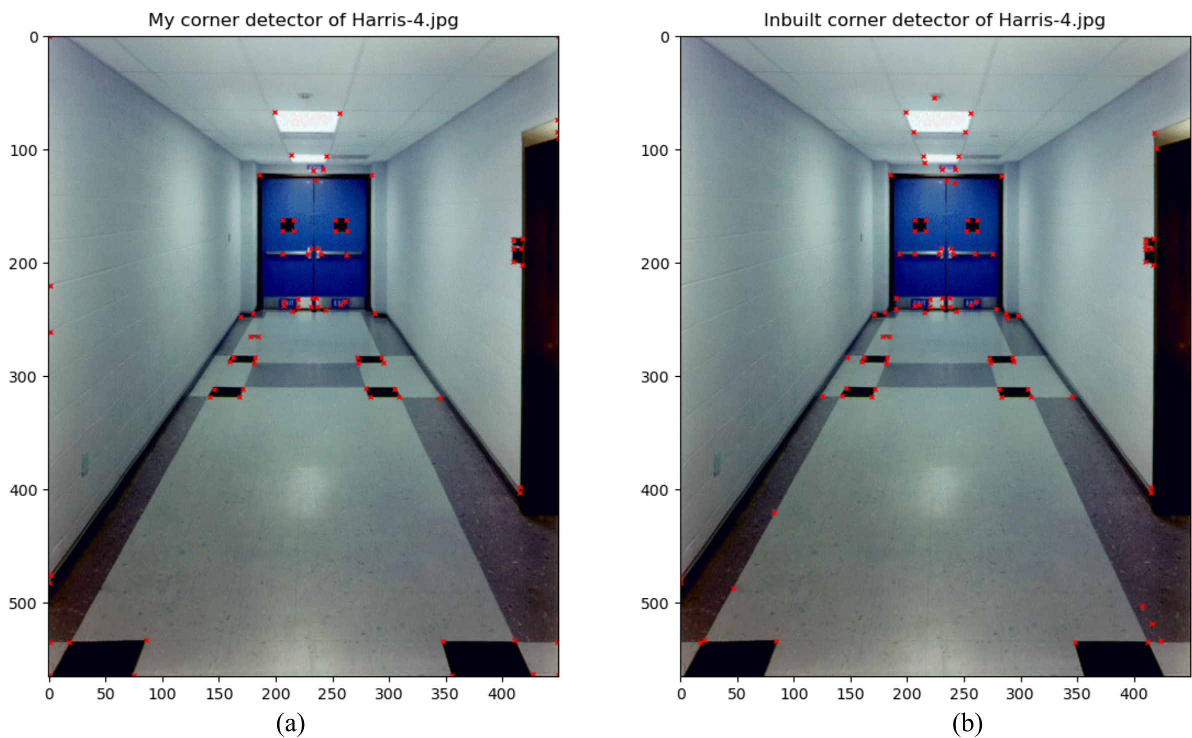
Apply both my own Harris corner detector and inbuilt Harris corner detector function on "Harris-3.jpg". The parameters for my detector are the same as described in the last question. For inbuilt function, block size is 3, k size is 3, empirical constant k is 0.04, threshold is 0.005 and window for maximum suppression is 9x9. Results are shown in 1.9.




(a) (b)

Figure 1.9 Detected corners of “Harris-3.jpg”. (a) My Harris corner detector (b) Inbuilt Harris corner detector

Apply both my own Harris corner detector and inbuilt Harris corner detector function on “Harris-4.jpg”. The parameters for my detector are the same as described in the last question. For inbuilt function, block size is 3, k size is 3, empirical constant k is 0.04, threshold is 0.005 and window for maximum suppression is 7x7. Results are shown in 1.10.




(a) (b)

Figure 1.10 Detected corners of “Harris-4.jpg”. (a) My Harris corner detector (b) Inbuilt Harris corner detector

A common method for identifying corners or other interest locations in digital photographs is Harris corner detection. Harris corner detection's effectiveness is influenced by a number of things. Below we discuss several factors that might affect the performance.

(a) Threshold Value

The Harris corner detection's accuracy and dependability are significantly influenced by the threshold value. Which pixels are regarded as corners depends on the threshold value. Numerous pixels that are not corners may be mistakenly categorised as corners if the threshold value is set too low. On the other side, some real corner pixels can be missed if the threshold value is set too high. For the Harris corner detector to operate at its best, choosing a suitable threshold value is crucial.

(b) Corner Orientation

The image's corner orientation can have an impact on how well the Harris corner detector performs. The foundation of Harris corner recognition is the idea that corners have a distinct gradient change in all directions. However, the Harris corner detector might not be able to detect corners effectively if they are oriented a certain way in the image, like in textured areas. To improve performance in certain situations, additional corner detection methods may need to be applied in addition to Harris corner detection.

(c) The Contrast in the Image

The Harris corner detector's ability to detect corners accurately is influenced by the contrast in the image. It could be challenging to discern between the corners and the surrounding areas if the image has low contrast. The Harris corner detector's performance may suffer as a result. In contrast, if the image has strong contrast, the Harris corner detector could over- or under-detect real corners. Therefore, before using the Harris corner detector, it is crucial to set the picture contrast to the proper level.

6. **Test your built function on 'Harris-5.jpg' (Harris-5.jpg can be downloaded from wattle.). Analyse the results why we cannot get corners by discussing and visualising your corner response scores of the image. (0.5mark)**

Apply my Harris Corner Detector on Harris-5.jpg, with threshold ratio 0.01, maximum suppress size 5x5 and empirical constant k 0.04. Result is shown in Figure 1.11.

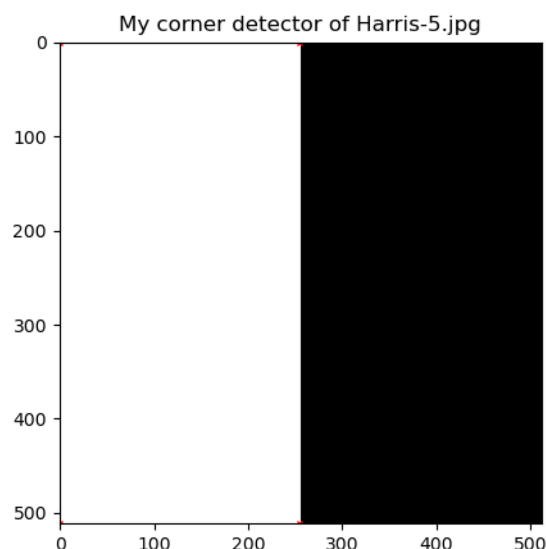



Figure 1.11 Detected corners of "Harris-5.jpg"

In order to analyze why we cannot get corners in this image, we need to analyze the R-value calculated before. Figure 1.12 shows a histogram of R-value for Harris-5.jpg.

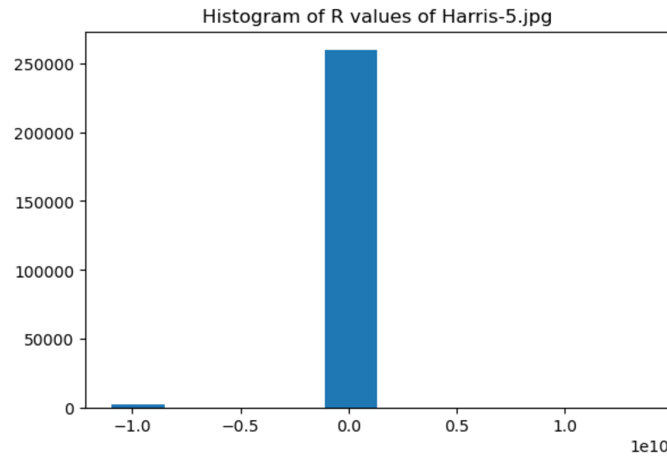



Figure 1.12 Histogram of R-value for "Harris-5.jpg"

In this histogram, we can see that most of R-values are close to 0, i.e. the absolute value is very small (The scale for the x-coordinate is 1e10), which means most of the area is flat. There are still some R-values which is less than 0, which indicates the edge. There are almost no R-values which is relatively high, indicating that there are no corner points in this image. This result corresponds to our image. We can observe Harris-5.jpg visually. Most of the area is rather black or white, indicating the "flat" area, with very small absolute R-values. And between black and white area, there exists an "edge", with R-values less than 0. There is no corner in this image, corresponding to that there are no large R-values.

7. **Test your built function on 'Harris-6.jpg' (Harris-6.jpg can be downloaded from wattle.). Plot your harris corner detector results in your report and propose a solution to obtain the salient corners which is robust to noise. (0.5mark)**

Apply my Harris Corner Detector on Harris-6.jpg, with threshold ratio 0.5, maximum suppress size 5x5 and empirical constant k 0.04. Result is shown in Figure 1.13.

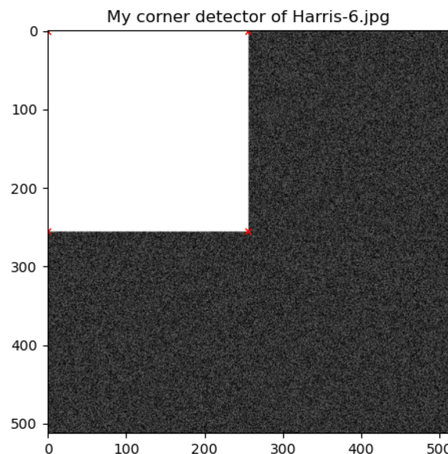
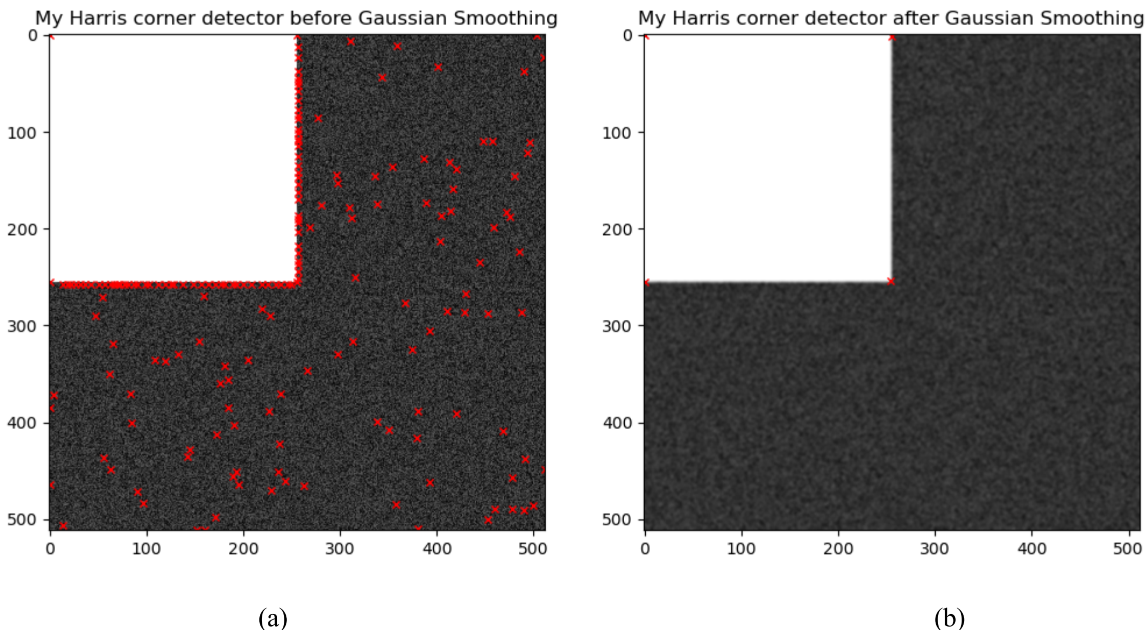



Figure 1.13 Detected corners of "Harris-6.jpg"

We can use methods like Gaussian smoothing to lessen the effect of noise on the image and increase the robustness of this approach to noise. A Gaussian kernel is convolved with the image during the process of Gaussian smoothing, which blurs the image and lessens high-frequency noise. Below Figure 1.14 is an example of using Harris Corner Detector with threshold 0.01, maximum suppress size 5x5 and empirical constant k 0.04 before and after using Gaussian smoothing method.




(a) (b)

Figure 1.14 Detected corners of “Harris-6.jpg”. (a) Using my Harris corner detector before applying Gaussian Smoothing method on the image (b) Using my Harris corner detector after applying Gaussian Smoothing method on the image

Both images are using the same threshold, maximum suppress window size, and empirical constant k. We can find that after applying the noise reduction technique, the Harris Corner Detector algorithm can be applied to the smoothed image to identify the salient corners. These corners can then be further refined and filtered to get a robust result.

# Task 2 - Deep Learning Classification (10 marks)

**In this lab, we will train a CNN with the Fashion-MNIST using the PyTorch[1] deep learning frame- work. The Fashion-MNIST dataset contains 70000 images: 60000 training images and validation images, 10000 testing images[2] for fashion classification task. Note that we have 60000 training images with labels. You are required to split the 60k training data further by yourself to reserve 1000 images for validation (they should be from 10 classes). Images are of size 28×28 Greyscale, for 10 classes:**

| Label | Description |
|---|---|
| 0 | T-shirt/top |
| 1 | Trouser |
| 2 | Pullover |
| 3 | Dress |
| 4 | Coat |
| 5 | Sandal |
| 6 | Shirt |
| 7 | Sneaker |
| 8 | Bag |
| 9 | Ankle boot |

1. **Download the Fashion-MNIST dataset from the following git link**

2. **After loading the data using numpy, normalize the data to the range between (-1, 1). Also perform the following data augmentation when training (1 mark):**
   - **randomly flip the image left and right.**
   - **zero-pad 4 pixels on each side of the input image and randomly crop 28x28 as input to the network.**

We can use transforms.ToTensor() and transforms.Normalize() to normalize the data to the range between (-1, 1). Besides, we can use transforms.RandomHorizontalFlip() to randomly flip the image left and right. First, we create a transforms.Compose object and then pass it to a torch.utils.data.Dataset object, as code shown in Figure 2.1.

```
# Define data transformations
train_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, ), (0.5, )),
    transforms.RandomHorizontalFlip(0.5),
])

# Load data with transformations
train_dataset = CustomDataset('./Data', 'train', transform=train_transform)
```

Figure 2.1 Normalize the data to the range between (-1, 1)

Define a transforms.Compose object that includes three transformations: transforms.ToTensor(), which converts the input data to a PyTorch tensor, transforms.Normalize(), which normalizes the tensor values to the range between (-1, 1), and transforms.RandomHorizontalFlip(), which randomly flips the image left and right.

The Normalize() transformation takes two arguments: the mean and standard deviation of the tensor values in the dataset. In this case, we set the mean and standard deviation to 0.5 for each channel.

The RandomHorizontalFlip() takes one argument: the possibility for an image to flip horizontally. We choose 0.5 as an input, which means there are 50% of the possibility for an image to flip.

Finally, we load the MNIST dataset using our own CustomDataset class and pass the transform argument to apply the defined transformations to the dataset.

We also need to add zero padding to the image and randomly crop 28x28 as input to the network. We choose torch.nn.functional.pad() to add padding (import torch.nn.functional as F), and we use transforms.functional.crop() to crop the image. Code is shown in the Figure 2.2, and these lines are in the CustomDataset class.

```
"Add zero-padding of 4 pixels on each side"
image = F.pad(input=image, pad=(4, 4, 4, 4), mode='constant', value=0)

"Get the dimensions of the padded image"
_, h, w = image.shape
"Randomly crop a 28x28 region from the padded image"
crop_x = np.random.randint(0, w-28+1)
crop_y = np.random.randint(0, h-28+1)
image = transforms.functional.crop(image, crop_x, crop_y, 28, 28)
label = torch.tensor(label).long()
```

Figure 2.2 Add zero-padding and randomly crop

3. **Build a CNN with the following architecture (1 mark):**
    - **5×5 Convolutional Layer with 32 filters, stride 1 and padding 2.**
    - **ReLU Activation Layer.**
    - **2×2 Max Pooling Layer with a stride of 2.**
    - **3×3 Convolutional Layer with 64 filters, stride 1 and padding 1.**
    - **ReLU Activation Layer.**
    - **2×2 Max Pooling Layer with a stride of 2.**
    - **Fully-connected layer with 1024 output units.**
    - **ReLU Activation Layer.**
    - **Fully-connected layer with 10 output units.**

We write a MyModel to store our CNN network, shown in Figure 2.3.

```
# Define a Model
class MyModel(nn.Module):
    # inherit from nn.Module class
    def __init__(self):
        super().__init__()
        # 5×5 Convolutional Layer with 32 filters, stride 1 and padding 2
```

```
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5,
stride=1, padding=2)
        # ReLU Activation Layer
        self.relu1 = nn.ReLU()
        # 2×2 Max Pooling Layer with a stride of 2
        self.pool1 = nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        # 3×3 Convolutional Layer with 64 filters, stride 1 and padding 1
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64,
kernel_size=3, stride=1, padding=1)
        # ReLU Activation Layer
        self.relu2 = nn.ReLU()
        # 2×2 Max Pooling Layer with a stride of 2
        self.pool2 = nn.MaxPool2d(kernel_size=(2, 2), stride=2)
        # Fully-connected layer with 1024 output units
        self.fc1 = nn.Linear(in_features=64 * 7 * 7, out_features=1024)
        # ReLU Activation Layer
        self.relu3 = nn.ReLU()
        # Fully-connected layer with 10 output units
        self.fc2 = nn.Linear(in_features=1024, out_features=10)

    def forward(self, x):
        # x shape [batch_size, channel, height, width]  [B, 1, 28, 28]
        x = self.conv1(x)                # [B, 32, 28, 28]
        x = self.relu1(x)                # ReLU function
        x = self.pool1(x)                # [B, 32, 14, 14]
        x = self.conv2(x)                # [B, 64, 14, 14]
        x = self.relu2(x)                # ReLU function
        x = self.pool2(x)                # [B, 64, 7, 7]
        x = x.view(-1, 64 * 7 * 7)       # reshape x to be [batch_size,
features]
        x = self.fc1(x)                  # [B, 1024]
        x = self.relu3(x)                # ReLU function
        x = self.fc2(x)                  # [B, 10]
        return x
```

Figure 2.3 My CNN network

**4. Set up cross-entropy loss. (0.5 mark)**

Cross-entropy loss is a commonly used loss function in classification problems. It measures the difference between the predicted probability distribution and the actual probability distribution. In Python, we can implement cross-entropy loss as shown in Figure 2.4.

```
import torch.nn as nn
loss_fn = nn.CrossEntropyLoss()
```

Figure 2.4 Set up cross-entropy loss

**5. Set up Adam optimizer, with 1e-3 learning rate and betas=(0.9, 0.999). (0.5 mark)**

Code is shown in Figure 2.5.

```
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
betas=(0.9, 0.999))
```

Figure 2.5 Set up Adam optimizer

Here, model refers to the CNN model that was defined earlier. The Adam optimizer is created using the optim.Adam function from PyTorch's optim module, and it takes three arguments:

(a) model.parameters(): This specifies the parameters of the model that need to be optimized. The parameters() function returns an iterable of all the parameters in the model.
(b) lr=1e-3: This sets the learning rate of the optimizer to 1e-3.
(c) betas=(0.9, 0.999): This sets the values of the beta parameters used by the optimizer. In this case, the first beta parameter is set to 0.9 and the second beta parameter is set to 0.999.

6. **Train your model. Draw the following plots:**
    - **Training loss vs. epochs.**
    - **Training accuracy vs. epochs.**
    - **Validation loss vs. epochs.**
    - **Validation accuracy vs. epochs.**

    **You can either use Tensorboard to draw the plots or you can save the data (e.g. in a dictio- nary) then use Matplotlib to plot the curve. (2 marks)**

We use Tensorboard to draw the plots. After training the model for 30 epochs, with Batch size 128 and Learning rate 0.001, plots of training/validation loss/accuracy are shown in Figure 2.6.

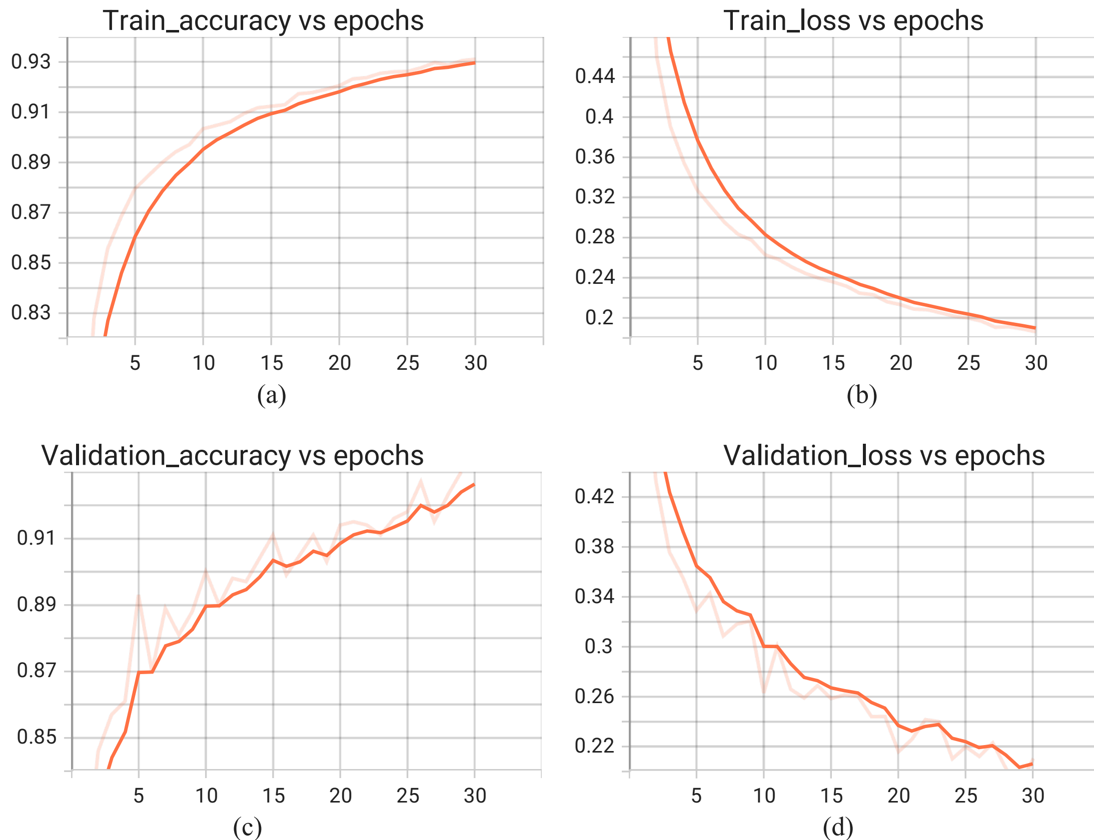


Figure 2.6 Results after training model. (a) plot of train_accuracy vs epochs (b) plot of train_loss vs epochs (c) plot of train_accuracy vs epochs (d) plot of train_loss vs epochs

And after training for 30 epochs with batch 128 and learning rate 0.001, test accuracy is 91.84, as shown in Figure 2.7.

```
 epoch 30 loss: 0.1748
Test Error:
   Accuracy: 93.60, Avg loss: 0.1729

Finished Training.
Test Error:
   Accuracy: 91.84, Avg loss: 0.2339


Process finished with exit code 0
```

Figure 2.7 Results for test data set

7. **Train a good model. Marks will be awarded for high performance and high efficiency in training time and parameters (there may be a trade-off), good design, and your discussion. You are not allowed to use a pre-trained model, you should train the model yourself. You need to describe what exactly you did to the base model to improve your results in your report, and your motivation for your approach (no more that 1 page of text). Please include plots as above for training and validation loss and accuracy vs. epochs, as well as the final accuracy on the test set. Please submit the code and your trained model for this. Your performance will be verified. Please ensure you follow the test/train splits as provided. You also must show your training and validation accuracy vs. epochs for this model in your report. Note that you may be asked to run training of your model to demonstrate its training and performance. (4 marks)**

My new CNN model is designed as follows:

Part 1:

- 3×3 Convolutional Layer with 32 filters, stride 1 and padding 1
- Batch Norm layer
- ReLU Activation Layer
- 3×3 Convolutional Layer with 32 filters, stride 1 and padding 1
- Batch Norm layer
- ReLU Activation Layer
- 2×2 Max Pooling Layer with a stride of 2

Part 2:

- 3×3 Convolutional Layer with 64 filters, stride 1 and padding 1
- Batch Norm layer
- ReLU Activation Layer
- 3×3 Convolutional Layer with 64 filters, stride 1 and padding 1
- Batch Norm layer

- ReLU Activation Layer
- 2×2 Max Pooling Layer with a stride of 2

Part 3:

- 3×3 Convolutional Layer with 128 filters, stride 1 and padding 1
- Batch Norm layer
- ReLU Activation Layer
- 3×3 Convolutional Layer with 128 filters, stride 1 and padding 1
- Batch Norm layer
- ReLU Activation Layer
- 2×2 Max Pooling Layer with a stride of 2

Part 4:

- Fully-connected layer with 1024 output units
- ReLU Activation Layer
- Drop out layer
- Fully-connected layer with 10 output units

This model consists of four sets of convolutional layers, each followed by batch normalization and ReLU activation, and max pooling layers to downsample the feature maps. The final layers are fully connected layers with a ReLU activation and dropout to prevent overfitting. The output layer is a linear layer with 10 units, representing the number of classes in the classification task.

The motivation behind this architecture is to capture hierarchical features from the input image data, using convolutional and pooling layers to extract local features and reduce dimensionality, and fully connected layers to perform classification based on the extracted features. By including batch normalization and dropout, the model aims to improve generalization performance and prevent overfitting.

Codes for this model are shown in Figure 2.8.

```
class MyModel2(nn.Module):
    # inherit from nn.Module class
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Sequential(
            # 3×3 Convolutional Layer with 32 filters, stride 1 and padding 1
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(32),
            # ReLU Activation Layer
            nn.ReLU(),
            # 3×3 Convolutional Layer with 32 filters, stride 1 and padding 1
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(32),
            # ReLU Activation Layer
            nn.ReLU(),
            # 2×2 Max Pooling Layer with a stride of 2
            nn.MaxPool2d(kernel_size=2, stride=2)
```

```
        )
        self.layer2 = nn.Sequential(
            # 3×3 Convolutional Layer with 64 filters, stride 1 and padding 1
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(64),
            # ReLU Activation Layer
            nn.ReLU(),
            # 3×3 Convolutional Layer with 64 filters, stride 1 and padding 1
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(64),
            # ReLU Activation Layer
            nn.ReLU(),
            # 2×2 Max Pooling Layer with a stride of 2
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.layer3 = nn.Sequential(
            # 3×3 Convolutional Layer with 128 filters, stride 1 and padding
1
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(128),
            # ReLU Activation Layer
            nn.ReLU(),
            # 3×3 Convolutional Layer with 128 filters, stride 1 and padding
1
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3,
stride=1, padding=1),
            # Batch Norm layer
            nn.BatchNorm2d(128),
            # ReLU Activation Layer
            nn.ReLU(),
            # 2×2 Max Pooling Layer with a stride of 2
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.fc1 = nn.Sequential(
            # Fully-connected layer with 1024 output units
            nn.Linear(in_features=128*3*3, out_features=1024),
            # ReLU Activation Layer
            nn.ReLU(),
            # Drop out layer
            nn.Dropout(0.5)
        )
        # Fully-connected layer with 10 output units
        self.fc2 = nn.Linear(in_features=1024, out_features=10)

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
```

```
        x = self.fc2(x)
        return x
```

Figure 2.8 New CNN model

Plots of training/validation loss/accuracy are shown in Figure 2.9.

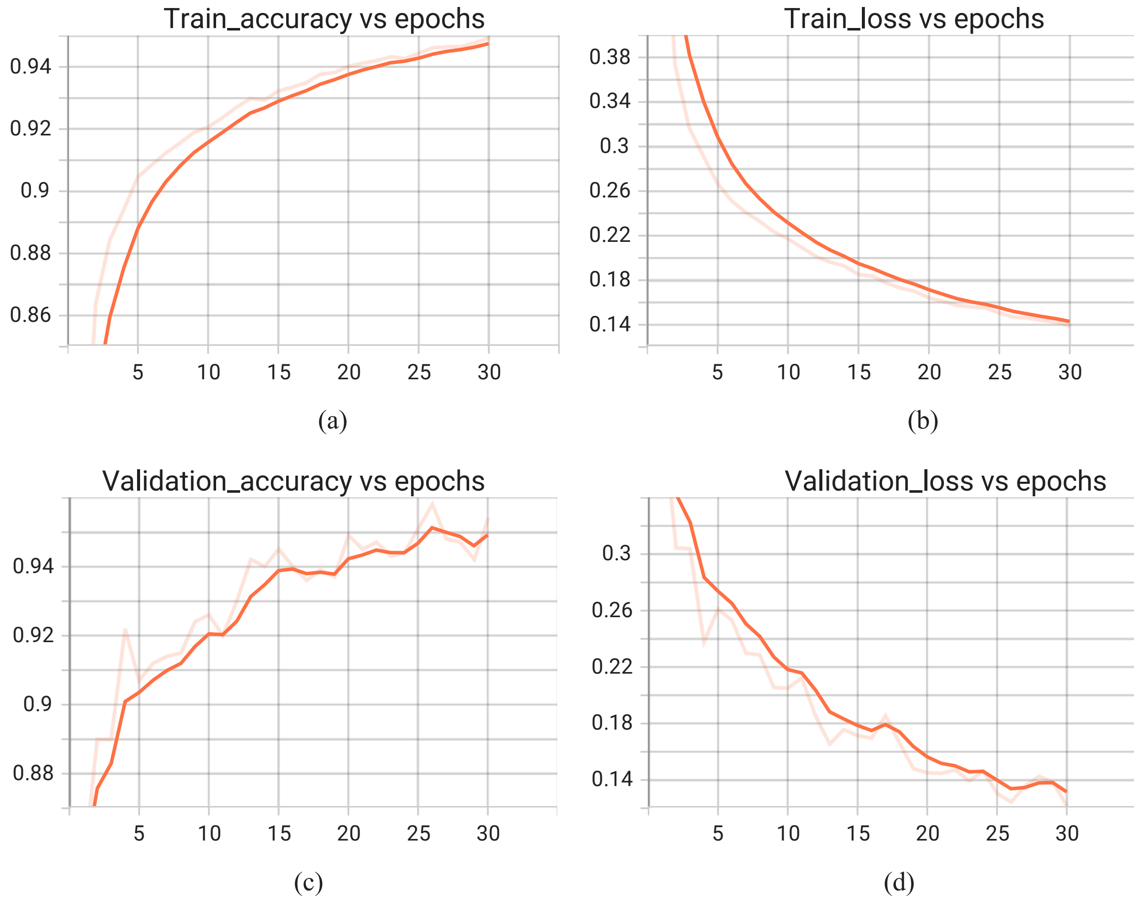



Figure 2.9 Results after training model. (a) plot of train_accuracy vs epochs (b) plot of train_loss vs epochs (c) plot of train_accuracy vs epochs (d) plot of train_loss vs epochs

And after training for 30 epochs with batch 128 and learning rate 0.001, test accuracy is 92.48, as shown in Figure 2.10.

```
 epoch 30 loss: 0.1418
Test Error:
   Accuracy: 95.40, Avg loss: 0.1216

Finished Training.
Test Error:
   Accuracy: 92.48, Avg loss: 0.2180


Process finished with exit code 0
```

Figure 2.7 Results for test data set

8. **The main dataset site on github (https://github.com/zalandoresearch/fashion-mnist) in- cludes a series of results for other network models (under Benchmark). How does your model compare? Explain why the ResNet18 model may produce better results than yours (or the other way around if this is the case).(1 mark).**

Based on the benchmark results provided in the GitHub repository, my model's accuracy of 92.48% on the Fashion-MNIST dataset is quite good and is comparable to the performance of some of the models on this dataset. For example, the MLP GRU+SVM with dropout model achieved an accuracy of 89.7%, while the Dyra-Net model achieved an accuracy of 90.6%. My model's accuracy is higher than these models.

However, my model's accuracy is lower than some of the other models, such as the VGG16 26M parameters model which achieved an accuracy of 93.5% and the WRN40-4 8.9M params model which achieved an accuracy of 96.7%.

There may be room for improvement and may consider trying different model architectures, hyperparameters, or training techniques to see if my model's accuracy can be further improved.

ResNet18 is a deep convolutional neural network model that has shown superior performance in image classification tasks compared to other models. Here are a few reasons why ResNet18 may produce better results than my model:

(a) Large datasets are frequently used to train ResNet18, which helps the network's weights get off to a solid start. The performance of this pre-trained model can then be enhanced using additional datasets.

(b) ResNet18 can learn more complicated features and patterns in the data because it has more layers and filters than my model does.

(c) ResNet18 may skip connections to add a layer's input to a later layer's output, enabling the network to skip some layers while maintaining crucial data from prior layers. This can help to reduce the degradation problem that can occur in deep networks.